Catching Accurate Profiles in Hardware

Satish Narayanasamy Timothy Sherwood Suleyman Sair
Brad Calder George Varghese

Department of Computer Science and Engineering
University of California, San Diego
{satish,sherwood,ssair,calder,varghese}@cs.ucsd.edu

Abstract

Run-time optimization is one of the most important ways of getting performance out of modern processors. Techniques such as prefetching, trace caching, memory disambiguation etc., are all based upon the principle of observation followed by adaptation, and all make use of some sort of profile information gathered at run-time. Programs are very complex, and the real trick in generating useful run-time profiles is sifting through all the unimportant and infrequently occurring events to find those that are important enough to warrant optimization.

In this paper, we present the Multi-Hash *architecture to catch important events even in the presence of extensive noise. Multi-hash uses a small amount of area, between 7 to 16 Kilobytes, to accurately capture these important events in hardware, without requiring any software support. This is achieved using multiple hash tables for the filtering, and interval-based profiling to help identify how important an event is in relationship to all the other events. We evaluate our design for value and edge profiling, and show that over a set of benchmarks, we get an average error less than 1%.*

1. Introduction

In traditional systems, software alone is used to gather program behavior information, either statically through binary instrumentation tools [18], or dynamically through just-in-time compilation [10]. Recently an area of active research has been the architectural support of generating profiles at run-time [15, 8, 11, 12, 5]. These prior techniques, while very effective at assisting software based profiling, are dependent on the system software for management or the aggregation of events. We present a **hardware-only** profiler that requires no support or knowledge of the overlying software layers. This will allow the profiler to be used by any internal hardware run-time optimization such as trace caching or prefetching regardless of the operating system type or version.

The goal of this paper is to design a profiling scheme that satisfies the following properties:

- **Area Efficient** - The amount of profiling events examined can create capacity constraints for a pure hardware profiler. A successful profiler must be able to deal with these capacity constraints gracefully using a fixed amount of area.
- **Accurate** - The number of times frequent profile events occur needs to be accurately counted. A hardware profiler must be able to sieve through all the noise and identify those events that are important enough to be used for optimization.
- **Timely** - The profiler should be able to provide up-to-date information about program behavior.
- **Performance Efficiency and Software Independent** - We would like a profiler that is not dependent on system software support to manage the profiles. Many proposed profilers use hardware to filter the events but then use software to accumulate and analyze them. Our goal is to avoid this dependence, and create an efficient technique for identifying these important events completely in hardware.

To design a pure hardware-based profiler that meets the above goals, we propose using (1) an interval-based profiler, and (2) a Multi-hash profiling architecture. Interval-based profiling breaks up execution into fixed intervals, and finds the frequently occurring events for each interval. By using a fixed interval, our profiler knows how many times an event needs to occur to be classified as an important event in relationship to all the other events. The *Multi-hash* architecture uses multiple hash tables to determine what events are important for a given profile interval. This can significantly increase accuracy in comparison to a single hash table given the same amount of hardware. We evaluate our hardware profiling architectures for value and edge profiling, and show that the Multi-hash architecture is able to attain an error rate of less than 1% on average.

The rest of this paper is organized as follows. Section 2 presents several potential uses of our profiling architecture. Section 3 describes the representation of profiling events used in this paper. In Section 4, prior work on profiling is discussed. Section 5 describes a single-hash table implementation of our profiler in more detail, simulation methodology, and how we calculate error. We extend this design to make use of multiple hash tables and this is presented in Section 6. Finally, we conclude in Section 7.

2. Motivation

There are many examples of hardware optimizations that can take advantage of information gathered at run-time. All of these following techniques can potentially benefit from the use of a more effective hardware profiling scheme.

Cache Replacement and Prefetching: In many cases a large percentage of data cache misses are caused by a very small

number of instructions. Two past techniques which have been shown to be effective at reducing memory latency are prefetching and speculative precomputation [4]. Making use of a run-time profiling scheme to identify troublesome loads and objects can potentially improve the accuracy and efficiency of these techniques.

Value based optimizations: Zhang et. al. found that in many programs about 50% of memory accesses are dominated by ten distinct values [22]. They use this information for storing compressed values in the data cache [21], but do not detail how those values can be captured dynamically. A hardware profiler could be used to capture this information.

Trace Formation: Another example of a hardware optimization based on run-time hardware profiling is trace formation. By dynamically extracting and ordering code that is frequently executed, instruction fetch can be made much more efficient [14]. In order to find the frequently executed code and to determine the best layout, a hardware profiling table is needed to track the run-time behavior.

Multiple Path Execution: One proposed optimization for branches that are hard to predict is Multiple Path Execution [19, 9, 20]. Multiple path execution tries to eliminate branch misprediction penalties by executing down multiple paths. While this technique can eliminate branch stalls, it comes at the price of increased execution resource demand. Therefore, this should not be done on all branches, only those that are known to be problematic. Finding these problematic branches is again a task that can be performed by a hardware profiler.

One common characteristic in all these optimizations is that they require information about events (e.g. loads that frequently miss in cache) that occur relatively frequently compared to the other individual events (e.g. other loads that rarely miss in the cache). By providing an accurate, efficient, and a fairly general approach for finding these events, we hope to extend or improve the applicability of the above mentioned optimizations.

3. Creating Unique Names for Profiling Events

The profiling events we target through hardware profiling may be a combination of several variables. This could include instruction PCs, load addresses, register values, register names, cache misses etc. To profile a given event we need to combine multiple variables into an identification (or name) that uniquely represents that event.

In this paper, motivated by the work by Sastry et. al. on Stratified Sampling [15], we concentrate on profiling events that require a combination of only two types of information. We define a *tuple* to be a pair of values. This pair of values uniquely identifies the event that is going to be given as input to the profiler. An example tuple used for load value profiling could consist of a $< LoadPC, value >$ pair. Similarly, for edge profiling $< branchPC, branchtargetPC >$ pair can be used.

Tuples are a flexible device for uniquely representing profiling events for many different types of optimizations like the ones described in Section 2, and further described in [15]. The techniques presented in this paper can be applied to non tuple based schemes, but by using tuples, it is very straightforward to label an event with the event identifier and the value of the event. This strategy will work well in a dedicated profiling environment where we can constrain the names to be those that are stored efficiently (such as a pair of addresses). If our profiling architecture is to be used in a generalized profiling engine, it can easily be extended to create unique names for events with multiple variables (more than two).

For the rest of the paper we will focus only on value profiling as done in [15]. In the end we also discuss results for branch edge profiling for our best profiling architecture.

4. Related Work

In this section we first give a classification of prior work on profiling. We then describe Stratified Sampling [15], which is the closest match to our profiler, in more detail.

4.1 Classification of Prior Profiling Techniques

The idea of run-time profiling has been around in different forms for many years. In order to understand how our generalized profiling architecture fits into the spectrum of related work, we classify the prior art into four categories and present examples from each classification. The traditional approach is to gather profiles in software and use it for static optimizations. A second approach is to assist software-based profiling with hardware counters. This idea was extended into the third technique, table based hardware profiling, which accumulates the events in the operating system software. The final class of profilers makes use of a separate profiling co-processor.

4.1.1 Software Profiling

Software based profiling is usually done with the assistance of a binary instrumentation tool such as ATOM [18]. One example technique for software based profiling is presented by Calder et. al. [3]. In [3], the authors instrument an executable using ATOM to capture the most frequently occurring values on a per-instruction basis. This information can then be exploited to perform value specialization [13].

4.1.2 Hardware Counter Assisted Profiling

Many modern processors include hardware counters for profiling. Software systems can then sample these counters to find information such as delay in cycles or number of cache misses. DCPI [1, 6] is an example of software system that takes advantage of the hardware counters in the Alpha processors via statistical sampling. The hardware counters are used to record counts and then generate an interrupt to software. Software is used to do random sampling and record information for later use. The DCPI framework was used by Burrows et. al. to perform Flexible Value Sampling [2, 6]. By combining the sampling of DCPI with an Alpha instruction set interpreter, they were able to profile several contiguous instructions at each random sample.

Hardware assisted profiling through counters is limited by a finite number of counters, and often requires time to collect many samples in order to develop a picture of the behavior to optimize. Hardware profiling via counters can be extended to the idea of using a hardware table of counters to more quickly and accurately gather a larger sample of data.

4.1.3 Hardware Table Based Profiling

Sastry et al. [15] proposed a hybrid profiling architecture called Stratified Sampling. This approach splits the input stream into disjoint substreams that are sampled independently. This results in the sampler converging to the desired

accuracies faster than a random sampler. We discuss this architecture in detail in Section 4.2.

Merten et al. [11, 12] developed a scheme for identifying program hot spots by profiling branch instruction execution frequency and history. It makes use of a hardware table to store this information on a per-branch basis. Conte et al. proposed a similar structure [5] for edge profiling. Their buffer also stores information only pertaining to branch histories, and is backed by memory. They investigate various indexing methods to increase profile accuracy across a range of table sizes.

Heil and Smith [8], propose the Relational Profiling Architecture which allows software to form queries regarding program behavior. These requests may either be about events on certain instructions or instructions that are being affected by a certain event. A hardware query engine then processes the queries, collects the desired information and passes it back to the software as messages. The query engine is a co-processor that executes the queries written in assembly language. Service threads read the messages containing profile information and perform optimizations using this data.

The above hardware-based table profiling techniques have to deal with error due to hardware capacity constraints, and incorporate custom replacement policies to try to reduce this error. In addition, these techniques aggregate their data in software to determine the important events, which creates overhead. For example, Sastry et al. [15] reports that their approach has a 5% software overhead when used for value profiling. In contrast, the hardware profiler we propose in this paper requires no software overhead to accurately capture the important events completely in hardware.

4.1.4 Co-processor profiler

Instead of including hardware tables for profiling, another option is to use a specialized co-processor for profiling and hardware-based optimization. Zilles and Sohi [23] show the use of a specialized profiling co-processor to distill information passed from the main processor. The main processor sets up the co-processor, which is mapped to a special address space. The co-processor can then filter instructions by a variety of means, and stores information into a buffer. Information is transfered from the co-processor to the main processor by either an explicit read from the processor, or by a co-processor generated interrupt which in turn backs the co-processors buffer to main memory.

As an alternative to dedicating a co-processor to hardware profiling, we show that our techniques can achieve very accurate profiling results completely in hardware using a small amount of storage (7 to 16 Kilobytes) at the cost of some flexibility.

4.2 Stratified Sampler

In this section, we analyze the stratified sampling technique proposed by Sastry et al. [15] for generic hardware profiling. Stratified periodic sampling divides the original input stream into multiple substreams via hashing. The events seen in a substream will have some correlation between them as a result of the hashing function applied on the original input stream. These substreams are then independently sampled using a conventional periodic or random sampler, one for every substream. This periodic or random sampler will experience less error rate as its input substream is biased. Consequently, the overall error rate of the stratified sampler will be less compared to having a single periodic or random sampler that takes the original stream as its input.

Figure 1 shows the design of the stratified sampler as proposed by Sastry et al. A table of counters is used to keep track of the number of occurrences of different events. A counter for an input event is selected by applying a hash function on the input event. This counter is incremented whenever the corresponding event appears in the input stream. When it reaches a sampling threshold value, it is reset and the event is reported to the profiling software by using an interrupt to the operating system.

To reduce aliasing and hence to improve accuracy, the authors propose adding partial tags and miss counters along with state information to the counter table. The hit counter keeps track of the number of occurrences of a tuple as in the simple design. The miss counter is incremented whenever a tuple hashes into that particular entry but its tag doesn't match with the one stored in that entry. This miss counter is used to guide the replacement policy. If too many misses occur to a particular tuple then the existing tuple is either evicted or discarded and replaced by the new one.

Note that in the above designs, every time a tuple reaches the threshold value, it is a candidate to be sent to the operating system. In order to reduce interrupt overheads, these messages are buffered and the OS is interrupted whenever the buffer fills up (100 entries in their study). In order to reduce the number of outgoing messages (and hence reduce overheads further), the authors examined placing a small fully-associative counter table next to the stratified sampler (and before the buffer) to aggregate information before sending it to software. Instead of immediately reporting a tuple that crosses the threshold value, they place it in this small associative table. If a counter in the associative table reaches a particular value or if an entry has to be replaced due to capacity issues then the event is passed on to software (via the intermediate buffer).

The proposed purpose of the Stratified Sampler is to collect behavior by accumulating it in the profiling software/operating system. Their results showed a 5% interrupt overhead for value profiling. In contrast, the goal of our architecture is to accumulate accurate profiles completely in hardware catching the frequently occurring events.

5. Interval-based Profiling for a Single Hash Profiler

In this section we describe the profiling architecture that enables us to accurately capture the frequently occurring events in hardware. Our single hash-table architecture is depicted in Figure 2. We start by modifying the stratified sampling architecture of Sastry et al. [15] shown in Figure 1. The first step is to remove the software feedback and to replace this with an accumulator table to completely capture the important profiling events in hardware. To be able to find temporal profiling information and to deal with the capacity issues caused by a finite amount of table space, we use interval-based profiling for our architecture. This allows us to identify when an event has occurred enough, within a given interval, to be classified as an important event. Using intervals requires the single hash-table architecture to reset all of the hash-table counters after every interval. The final modification to improve the accuracy

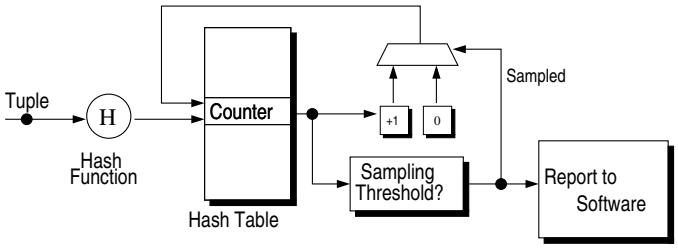

Figure 1: Stratified Sampling Architecture. The hash function computes a signature for each event. The signature is used as an index to select a counter in the counter table. The selected counter is then incremented. If the counter reaches its maximum threshold value, the counter is reset to zero, and the event is reported to the profiling software.

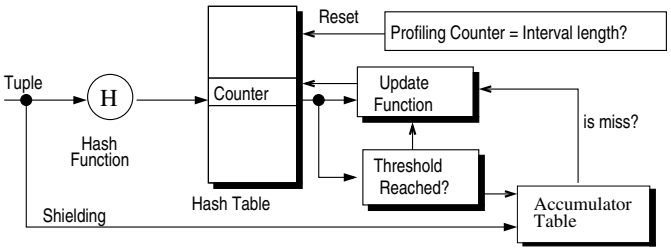

Figure 2: Single Hash Architecture. An input tuple is first checked in the accumulator table. If it is found, we only increment the counter in the accumulator table, and do not update the hash-table. If the event is not in the accumulator table, the hash function computes a signature for each event. The signature is used as an index to select a counter in the hash table. The update function is then applied to the selected counter. If the counter reaches its maximum threshold value, the event is moved into the accumulator table.

of our architecture is by using a technique we call shielding, which is described later. These three differences can be seen when comparing Figure 2 to Figure 1.

5.1 Interval Based Hardware Profiler

To meet the goals set forth, we make two important design decisions. The first decision is to divide the execution time into *intervals*. The interval length is a fixed number of profiling events (i.e. input tuples). The second decision is to capture only those tuples that occur for more than a predetermined percentage of interval length, which we call the *candidate threshold*. Tuples that occur for more than the candidate threshold during a given interval are called *candidate* tuples.

5.2 Single Hash Architecture

The accumulator table in Figure 2 is fully associative and has tags. An input tuple is first checked if it is in the accumulator table. If so, then we only increment the counter in the accumulator table, and do not update the hash-table. If the event is not in the accumulator table, then it is hashed into the hash-table and the corresponding counter is incremented. The hash-table contains a set of counters that keep track of the number of times an event occurs. This hash-table doesn't contain any tags and hence experiences problems due to aliasing.

Once the corresponding counter of a tuple reaches the candidate threshold value, an entry is allocated in the accumulator table. If there are no more free entries in the accumulator table for that profiling interval, then the event is not put into the accumulator table. When a tuple is inserted into the accumulator table it is marked as non-replaceable for the rest of the interval. In addition, that particular tuple will never be given as input to the hash-table for the rest of the interval to reduce the pressure on the hash table. We call this technique *shielding*. This is important to help reduce error rates, since it reduces the pressure on hash tables. At the end of an interval, the hash table is flushed. The accumulator table is not flushed, and instead all entries are marked as replaceable.

Profiling based on intervals and concentrating on capturing only the top occurring events allows us to have a very high probability that the accumulator table will not overflow during any given profile interval and thereby avoid capacity and aliasing issues. This allows us to calculate the worst case number of entries that might fill the table. Without this property, the accumulator table would have capacity problems and counters for a custom replacement policy would be needed (as proposed for many hardware structures to deal with capacity issues). For our interval-based architecture, there are two parameters (*profile interval length* and *candidate threshold*) used to figure out how large the accumulator table needs to be to make this guarantee.

The profile interval length is the number of profiling events that determine the duration of a profiling interval. At the end of the profile interval, the accumulator table contains the top candidate events, and this information can be used to optimize the behavior of the program during the next profile interval.

The candidate threshold is the percentage of occurrences that a particular event needs to occur, with respect to the profile interval length, in order for the event to make its way into the accumulator table. For example, one may specify that a particular branch is treated as a candidate only if it is responsible for 0.1% of all branch mispredictions. In this case the candidate threshold is 0.1%, and a branch is only moved into the accumulator table if its frequency for that profile interval is greater than or equal to 0.1% of the profile interval length.

Assuming a 10,000 interval length and making the architecture tailored towards only capturing the top 1% of most frequently occurring events, we only need an accumulator table of 100 entries to capture those events in the worst case. Similarly, if we are interested in the top 0.1% of events, we only use an accumulator table size of 1,000. This is key to providing a bound on the accumulator table size and to guarantee a low error rate. For this paper, we discuss results for short intervals of length 10,000 events and also for long intervals of length one million events. We concentrate on providing results for 10,000 events with 1% threshold, and also for configurations with 1 million events with 0.1% threshold. These two configurations allow us to examine the responsiveness and

the ability to quickly train our hardware profiler with a profile cycle of 10,000 events, and to examine severe pressure on our hash table, which has 2K entries, examining a profile cycle of 1 million events with a candidate threshold of 0.1%.

5.3 Hash Functions

For the hash table index, we use the same hash function as the one used for Stratified Sampler [15]. For a given tuple $< pc, value >$ the hash index is computed as follows. npc = flip(randomize(pc)); nv = randomize($value$); index = xor-fold(npc xor nv, index-size). The function *randomize* looks up for each byte of the input value a random number from a 256-entry random number table. It then composes these bytes together to create *nv* and the first part of the function for *npc*. The randomize function can be hardwired into the hash-table lookup. The function *flip(v)* reverses the bytes of *v*. *xor-fold(v,n)* splits *v* into chunks of n-bits and *xor*s those chunks to get the final value.

The intuition behind designing the above hash function is as follows. It can be the case that the *pc* addresses used near each other during execution only vary slightly, especially temporally close tuples examined during profiling. The same applies to the *value*s seen during execution. Therefore, the *randomize* function is used to magnify this small amount of variation. In addition, the *pc* and *value* may not vary significantly in their higher order bits. Hence, through *flip* operation, we move the variation in one member of the tuple, *pc*, to the higher order bytes. When this is xor-ed with *value* we are able to obtain a greater degree in variation. In examining histogram of the number of static tuples that hashed to each table entry, we found a very even distribution using the above hash function.

Multi-hash architecture to be discussed later in Section 6, requires many independent hash functions. We obtained such independent hash functions by just choosing different random number tables used by the function *randomize*.

5.4 Single Hash Table Optimizations

We now describe two optimizations to achieve lower error rates for a single hash table architecture.

5.4.1 Retaining

Retaining is used to keep the top candidates from the prior interval in the accumulator table at the start of the next interval. At the end of an interval, we flush all entries in the accumulator table that were not above the candidate threshold. We then mark all entries that had a value above the threshold as replaceable and set their counter values to 0. Therefore, in the next interval if these same set of tuples emerge as candidates, retaining will help reduce contention for the hash tables because of the use of our shielding optimization. When the value in the accumulator for an entry marked as replaceable (due to retaining) goes above the candidate threshold, then it is unmarked as replaceable for the rest of that interval. For allocating accumulator entries, empty entries are allocated first followed by replaceable entries.

5.4.2 Resetting

The goal of this optimization is to reduce the number of false positives. The counter in the hash-table is reset after a profiling event reaches the candidate threshold, and is promoted to the accumulator table. In doing this, we make sure that other events that alias to that entry will not migrate to the accumulator table. This can have the negative effect of increasing the error rate if we reset the hash table entry being shared by another event. This other event may then end up not making its way into the accumulator table when it should have.

5.5 Methodology

To perform our study, we collected information for three SPEC95 programs (`go`, `li` and `m88ksim`) and two SPEC 2000 programs (`gcc`, and `vortex`) for their reference input sets. Each program was compiled on a DEC Alpha AXP-21164 processor using the DEC C, C++, and FORTRAN compilers. The programs were built under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). In addition, we collected results for three C++ programs `deltablue`, `sis` and `burg`. `deltablue` is a constraint solution system, `sis` is a synthesis of synchronous and asynchronous circuits, and `burg` generates a fast tree parser using BURS technology.

We chose these programs for their large number of static instructions executed and their ability to exhibit capacity problems on profiling hardware structures. We performed all of our profiling analysis using ATOM [18]. For all of the results, the programs were fast forwarded using the fast forward numbers from SimPoint [16, 17], and then ran for 500 million instructions.

5.5.1 Error Calculation

To calculate the error of the profiling architectures, we focus on how well during a given interval the profiler can capture the candidate tuples. For each interval, we compare the candidates captured by our profiler to the candidates seen by a perfect profiler. Comparing the *hardware profile* to a perfect profile leads to four categories of errors as shown in Figure 3. They are:

- *False Positive* - Events that are identified as candidates by the hardware profiler but not by the perfect profiler. When a profile has too many false positives then it may trigger over aggressive optimizations. This may lead to severe degradation in performance especially if the misclassification is high.
- *False Negative* - Events that are not captured by the hardware profiler but are actually identified as candidates by the perfect profiler. Error due to false negatives will result in missing out potential opportunities for optimization.
- *Neutral Positive* - Profiling events that are identified as candidates by the hardware profiler as well as by the perfect profiler. Error occurs when the number of occurrences of a candidate in the hardware profiler is more than the number of occurrences in the perfect profiler.
- *Neutral Negative* - Same as Neutral Positive except that the frequency of a candidate reported by the hardware profiler is less than what is seen in the perfect profiler.

5.5.2 Calculating Error Rate

For a given profiler we calculate the error rate in the following way. Let n be the total number of candidates seen in

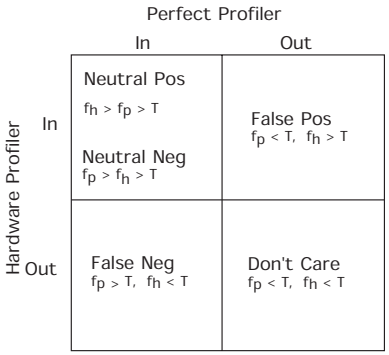

Figure 3: Metrics defined. For a tuple, *In* means it is identified as a candidate by the profiler. *Out* means it was found to be below the candidate threshold by the profiler. f_p, f_h are frequencies as seen in the perfect and hardware profile respectively. T is the candidate threshold frequency.

an interval in either the perfect or hardware profiler. Let f_{p_i} and f_{h_i} be the frequencies as seen by the perfect profiler and the hardware profiler respectively. If a candidate is a false negative, then its tuple will not be found in the accumulator table of the hardware profiler, and its f_{h_i} will be 0. The error rate E_i for a candidate i is calculated as $\frac{|f_{p_i} - f_{h_i}|}{f_{p_i}}$. The total error rate E for an interval is then calculated as the weighted average over error rates of all candidate tuples seen either in perfect or hardware profiler as shown in formula 1.

$$E = \frac{\sum_{i=1}^{n} \frac{|f_{p_i} - f_{h_i}|}{f_{p_i}} * f_{p_i}}{\sum_{i=1}^{n} f_{p_i}} \quad (1)$$

The final net error rate is calculated as a simple average over the error rates seen by all intervals. In Figure 3, we show the relationships of f_p, f_h, and the candidate threshold that are satisfied when classifying a tuple into one of the four categories. In the results we will show error rates split into Neutral Positive, Neutral Negative, False Positive, and False Negative.

5.6 Single Hash Table Results

We now present results using the single hash table for value profiling examining the number of candidate tuples found and the error rates.

5.6.1 Analysis of Candidate Tuples

It should be noted here that the accuracy of hardware profiling will depend primarily on (1) the number of unique tuples seen in an interval, which we call distinct tuples, and (2) the number of unique candidate tuples that crossed the threshold for a given interval. This section quantifies these aspects.

Figure 4 shows the number of distinct tuples seen on average in an interval for value profiling. The tuple for value profiling is represented as $< pc, value >$. Results are shown for ten thousand, one hundred thousand and one million interval lengths on a logarithmic scale. Figure 5 shows the number of unique candidate tuples that cross the threshold of 1% and 0.1%. These results are gathered using a perfect interval based profiler in order to show the *actual* number of distinct tuples

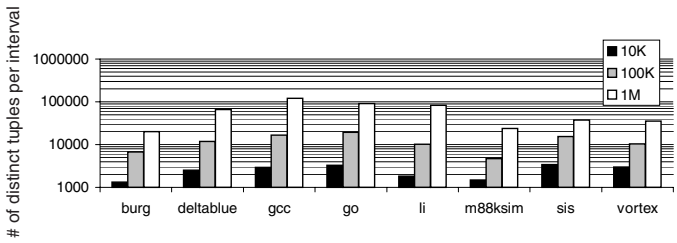

Figure 4: Number of distinct tuples seen in an interval on average. Results for different interval lengths are shown.

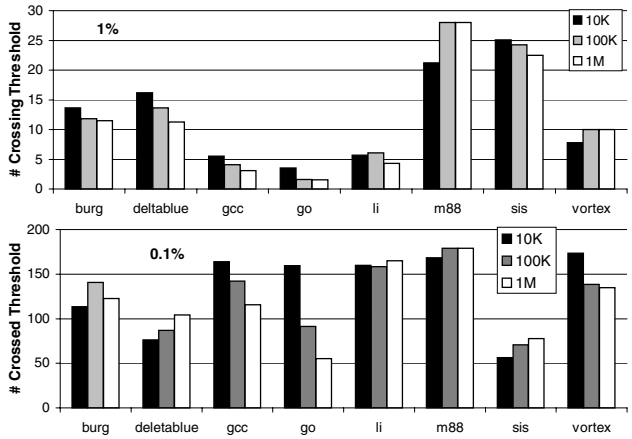

Figure 5: Number of unique candidate tuples in an interval on average. Results for different interval lengths are shown. The figure on the top is for a threshold of 1% while the figure on the bottom is for a threshold of 0.1%

and candidate tuples our profiler is going to encounter.

Figure 5 shows that the number of unique candidate tuples that cross the threshold is very small compared to the total number of distinct tuples shown in Figure 4. These static candidate tuples, though small in number, account for a large percentage of the total number of dynamic tuples seen during an interval of execution.

Another interesting result is that the total number of distinct tuples in an interval increases proportionally to interval length. Whereas, the number of unique candidate tuples that cross the threshold roughly remain the same irrespective of interval length. This implies that we have a tougher job in filtering out the candidate tuples as we increase the interval length because ratio of signal (those tuples we want to target) to noise (rarely repeating tuples) decreases.

One proposal for using a hardware profiler is to use the accumulator table information gathered during one profile interval to optimize behavior in the next profile interval. For this to be meaningful, candidate tuples for the current interval should also be candidates in the next profiling interval. Results presented in Figure 6 shows the change in candidate tuples found in the accumulator table between two consecutive profile intervals. Each point (x, y) in the graph shows that x% of intervals experience less than y% change in candidate tuples. For example, for one million interval length,

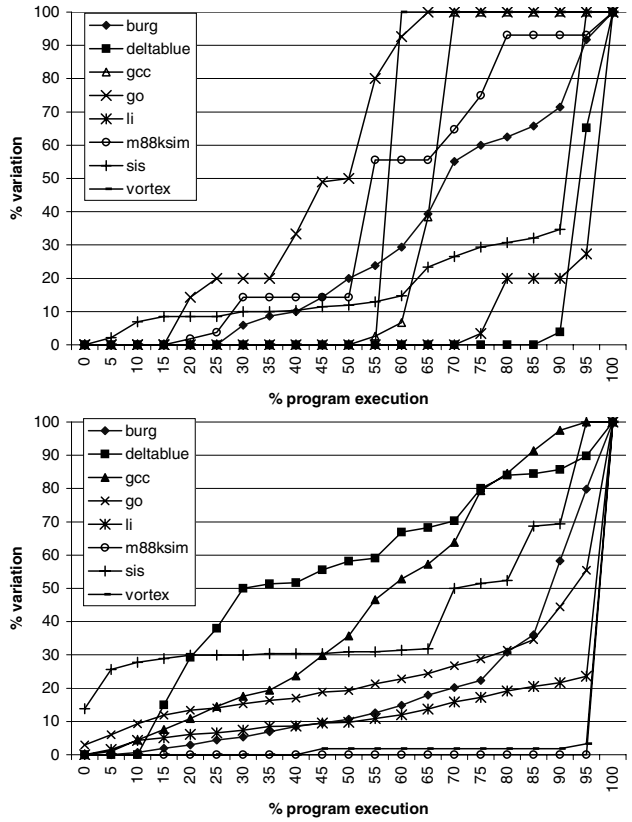

Figure 6: Percentage of variation of candidates from one interval to the next. The figure on the top is for an interval length of 10,000 events and the one on the bottom is for 1 million.

`gcc` experiences less than 35% variation for 50% of execution. The graph on the top is for an interval length of 10,000 with 1% threshold, while the one on the bottom is for 1 million interval length with 0.1% threshold.

The results show that for a profile interval of 1 million, `m88ksim` and `vortex` have very little variation, whereas they have a larger degree of variation for a profile interval of 10,000. For these programs, an interval of 10,000 events is too small to see all of the important tuples that occur temporally across the intervals of execution, whereas they are accurately captured with an interval of 1 million. In contrast, programs like `deltablue` have a higher variation for a 1 million interval, indicating that `deltablue` has large scale differing behavior as it executes through its different phases of execution. In comparison, for a 10,000 length interval, `deltablue` has very little variation of candidate tuples between intervals for most of its execution.

These results show that different interval lengths suit different programs, and it is important to pick the appropriate interval length to better capture a program's behavior. To address this, one can potentially adaptively pick the appropriate interval length for a given program.

5.6.2 Error Rates

Figure 7 shows the value profiling error rates for our single hash table architecture for four configurations. These are for the four different combinations of using or not using *resetting* (R) and *retaining* (P) optimizations. R0 means that no reset optimization was used, while R1 means reset was used. Similarly, P0 means retaining was not used, whereas P1 means retaining was used.

The results show that both resetting and retaining, decrease the total amount of error, and applying both optimizations performs the best overall. Resetting reduces the number of false positives and the total error significantly for the reasons discussed above, but it comes at the cost of an increase in false negatives when profiling some programs such as `vortex`. Across intervals, the *retaining* optimization reveals its' usefulness by significantly lowering the total number of false positives for most of the programs. This is even more drastic in the graph showing the results for an interval of one million and threshold of 0.1% (shown on the right), especially when the scale of the y-axis is considered.

6. Multi-hash Profiler

In this section we present and analyze a new hardware profiling architecture which we term the *multi-hash* design. This design is based on the architecture proposed by Estan and Varghese [7] for measuring traffic in network processors. In a network, one would like to account for the amount of bandwidth consumed by those users that are taking the most. In [7], the problem is to find a tight *lower bound* on the amount of resources consumed by those users taking up a certain amount of bandwidth or more.

6.1 Architecture Design

Our multi-hash architecture is built upon the simple interval based single hash hardware profiler described in the previous section. The multi-hash architecture uses multiple hash tables to reduce the number of false positives. Two tuples that aliased to the same entry in a single hash architecture, will probabilistically map to different entries for one or more of the multiple hash tables. This is the key feature behind the multi-hash architectures ability to greatly reduce the number of false positives, which we explain in more detail below.

Figure 8 shows the architecture of the muti-hash profiler. It consists of multiple hash-tables of counters and an equal number of independent hash functions (i.e. one for each hash-table). An input tuple is first indexed into the accumulator table. If it doesn't have an entry in it, then it hashes into all the hash-tables using the hash function for each hash-table, and all the corresponding counters are incremented. An entry in the accumulator table is added for a tuple only when *all* of its' corresponding counters across the different hash-tables cross the threshold candidate value. As in the single hash approach, we have two options when the counters in all the hash-tables reach the threshold value. We can either reset all those counters immediately, or we can reset the counters at the end of an interval. We examine these tradeoffs below.

Another optimization we examine for the multi-hash architecture is the use of *Conservative Update* from [7]. This technique does not always increment the corresponding counters in all the hash-tables. Instead, only the counter that has the smallest value among all other corresponding counters for the tuple is incremented. If there is a tie between counters with the smallest value, then each one of them is incremented. The intuition behind this optimization is that there should only be

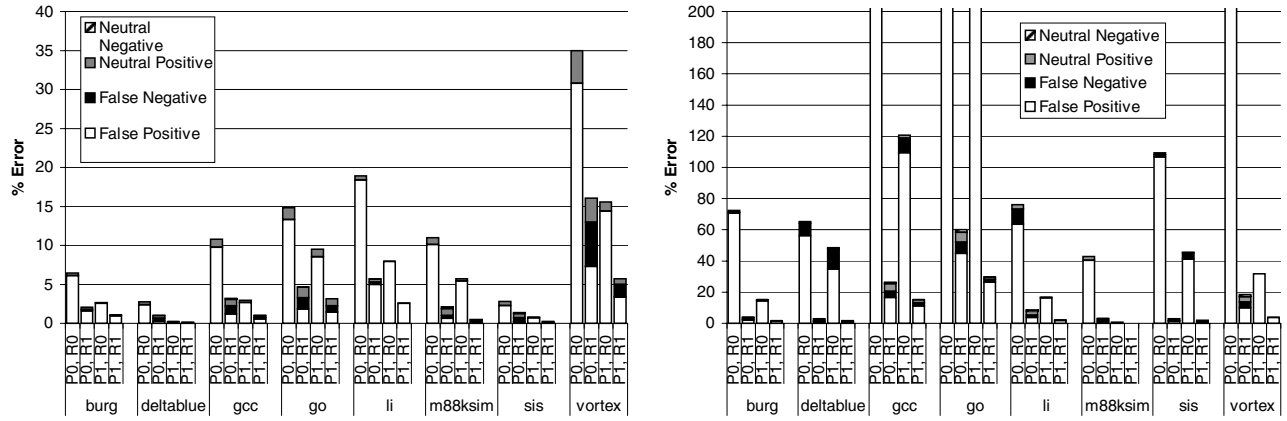

Figure 7: Single Hash table with retaining/resetting results across a set of benchmarks. R0 means that there is no reset optimization used for the accumulator table, while an R1 means that reset is enabled. A P1 means that retaining (or passing) is turned on, while a P0 means that there is no retention. Results are shown for intervals of 10K with a threshold of 1% on the left and an interval of 1 million with a threshold of 0.1% on the right.

a difference between the values in the presence of aliasing. In other words, when there is no aliasing, the counters in each hash-table should be identical. This optimization works by not incrementing counters that experience aliasing, which in turn can significantly reduce the amount of error.

6.2 Theoretical analysis

In this section we provide an *approximate* theoretical analysis for the multi-hash profiler to provide insight into why it works. We focus on finding the probability p for an input tuple being classified as a false positive. We do this analysis for configurations with different hash-table sizes and with different number of hash-tables.

We first describe the analysis for a single hash table. Let Z be the size of a hash-table and assume we want to capture the distinct tuples that occur for more than t% of interval length. Then there can be at most $100/t$ distinct tuples that occur for more than t% of any value. Note that the interval length has no impact on this bound. For an input tuple to be registered as a false positive, it needs to hash into a hash-table entry (counter) whose value is greater than or equal to the threshold value. The total number of such counters (above the threshold) is equal to $100/t$. There are Z distinct counters (where Z is the hash-table size). Therefore, the probability p for a tuple to turn into false positive is $100/(tZ)$.

Now, for a multi-hash configuration assume the total number of counters in a hash-table is actually Z/n where n is the number of hash-tables in multi-hash profiler. For a particular hash-table, the probability for a tuple to hit a counter and increment it to a value greater than threshold is $100/(tZ/n)$ which is $100*n/tZ$. This has to happen for all hash-tables for a tuple to turn into a false positive. Since the hash-tables are independent of each other, the final probability for an input tuple to turn into false positive is $(100*n/tZ)^n$. Figure 9 plots this function. Each curve represents a multi-hash configuration keeping the number of hash entries fixed, while varying the number of hash tables on the x-axis. For example, the 500 entry curve shows results for one 500 entry hash table, two 250 entry hash tables, and three 166 entry hash tables.

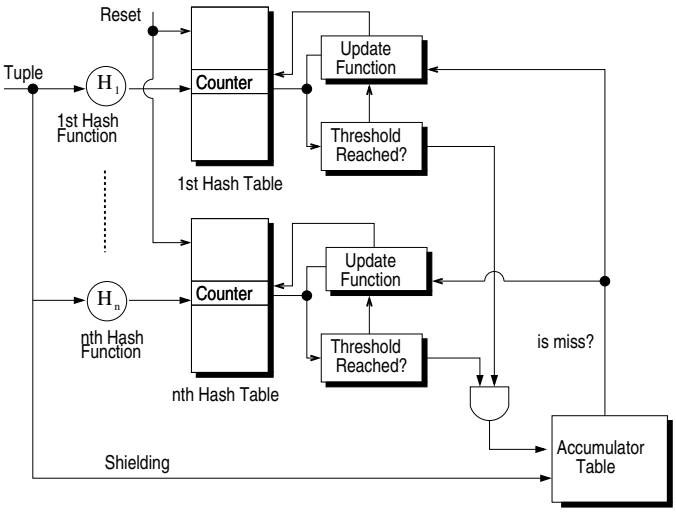

Figure 8: A multi-hash architecture. An input tuple is first checked in the accumulator table. If it is found, we only increment the counter in the accumulator table, and do not update the hash-table. If the event is not in the accumulator table, each hash function computes a signature for the event. These signatures are used as an index to select a counter in the counter tables. The update function is then applied to each selected counter. Only after every respective counter reaches its maximum threshold value in every hash-table, the event is moved in to the accumulator table.

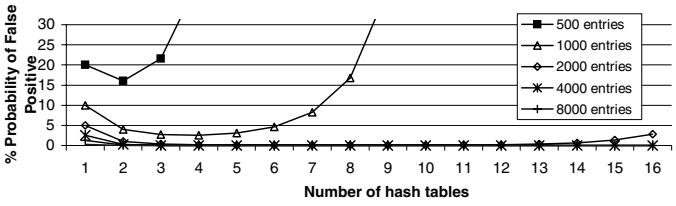

Figure 9: Theoretical analysis for a profiler with multiple hash-tables for 1% candidate threshold. This graph gives a theoretical upper bound on the probability for an input tuple to turn into a false positive. The X-axis shows the number of hash-tables. Each line represents a multi-hash configuration where the number of hash entries specified are split evenly between that number of hash-tables.

The results show that given a fixed amount of entries, increasing the number of hash-tables reduces the number of false positives up to a point, where performance starts to degrade due to aliasing problems. This can be seen for 1,000 entries, where performance degrades beyond 4 hash tables. Note that these probabilities provide a loose upper bound on the false positive error rates. The analysis does not take into account the number of unique tuples in an interval, the distribution of candidate tuples and non-candidate tuples over an interval, nor optimizations like retaining, shielding and conservative update. These would need to be taken into consideration to provide a tighter bound.

6.3 Design space analysis

In this section we analyze the different configurations that are possible for a multi-hash profiler and show results for a combined hash-table size of 2K entries. We performed error rate analysis for other hash-table sizes and found that a hash-table of size 2K performs almost as well as larger hash-tables, while still outperforming hash-tables of size 1K or smaller. We don't report the error rates for hash-table sizes other than 2K due to lack of space. It should be noted that all comparisons between multi-hash architectures with different numbers of tables have the same number of total hash-table entries (2K entries). These entries are evenly divided among the hash-tables. So a multi-hash profiler with n hash-tables will have $2K/n$ entries in each hash-table. We found that our approach of retaining tuples for the next immediate interval turned out to be useful for all configurations, and therefore we use this technique for all the results reported in this section.

We now examine the following options for a multi-hash profiler. We first vary the number of hash-tables, and examine the tradeoffs between using `reset` and `conservative update` for the multi-hash architecture. We denote *R1* as a configuration with immediate reset, and *R0* as without immediate reset. We denote *C1* as a configuration using conservative update, and *C0* as without. Figure 10 shows the results for an interval of length 10,000 with 1% candidate threshold, while Figure 11 shows results for an interval of length 1 million with 0.1% candidate threshold. In all configurations, the values in the hash tables are set to zero at the end of a profile interval. We focus on the results for `gcc` and `go`, since they have the largest number of unique tuples.

Note that the use of immediate resetting increases the overall error rate due to the high number of false negatives. This problem is emphasized as the number of hash-tables increases. For example, an 8 hash-table multi-hash profiler examining the program `go` in Figure 10 has an additional error rate of about 4% due to false negatives alone over configurations that don't use immediate reset. This is because when a counter is reset, a tuple does not occur enough times afterwards to warrant promotion to the accumulator table.

Another observation is the effectiveness of the conservative update approach. For example, in Figure 11, the error rate on the program `go` is brought down to under 1% using the multi-hash architecture. It should be noted that even with immediate resetting the error rate without conservative update still remains around 100% or higher for `go` when using an interval of 1 million with 0.1% candidate threshold.

6.4 Results for best multi-hash configuration

In the previous section we concluded that a multi-hash profiler without immediate reset and with conservative update approach performs the best of all configurations. In this section, we present the results by varying number of hash-tables for this best configuration. Here again we use a multi-hash architecture with a total of 2K hash table counters with an accumulator table with retaining. Along with the multi-hash profiler results, we also include results for the best single hash (BSH) approach from Section 5 for comparison.

6.4.1 Value Profiler

Figure 12 shows the results for the best multi-hash configuration for an interval of length 10,000 with 1% candidate threshold and also for interval of length 1 million with 0.1% candidate threshold. It can be observed that the multi-hash profiler with 4 hash-tables consistently outperforms all other configurations including best single hash approach. For example, in the right graph of Figure 12, for `go`, the muti-hash with 4 hash-tables shows an improvement in error rate of about 10% (absolute) over best single hash approach. But when we increase the number of hash-tables beyond 4, the error rate starts increasing due to false positives. This is similar to the 1K entry theoretical results seen in Figure 9.

To summarize the results, the lowest error rates in Figure 12 for `gcc` (5%) and for `go` (1.5%) are achieved using the multi-hash architecture. This is even when we use an interval of length 1 million with 0.1% candidate threshold, where severe capacity pressure is placed on our multi-hash architecture from all of the distinct tuples. This is a significant improvement when compared to the lowest error rates of 10% and 20% for `gcc` and `go` using the single hash table approach. More importantly, the average error we see for the multi-hash architecture is under 1%.

Figure 13 shows the error rate for each interval across the execution of the benchmarks examined. It shows results for the error rate observed in each individual interval of execution using an interval length of 1 million with a 0.1% candidate threshold. In Figure 13, the graph on the left shows results for the best single hash profiler with resetting, and the graph on the right shows results for the multi-hash profiler with conservative update and no resetting with the entries evenly split over 4 hash-tables. It can be seen that for the best single hash profiler, error rates for `gcc` remain high in the first 60 intervals and remains very low towards the end. Our best multi-hash profiler brings down the magnitude as well as the

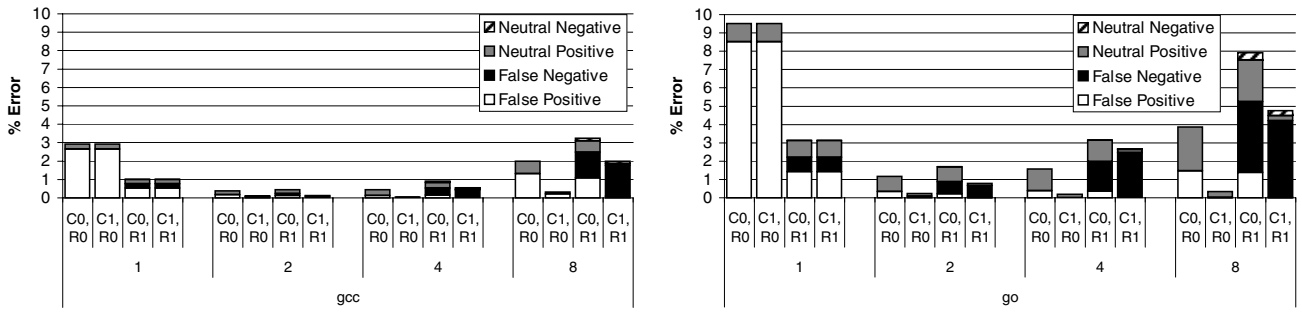

Figure 10: Muti-hash profiler for an interval of 10,000, 1% candidate threshold, and a total number of 2K hash-table entries. Each graph shows results for multi-hash profiler with 1,2,4 and 8 hash-tables. For each hash-table, it shows results for 4 different configurations: C0-R0, C1-R0, C0-R1, and C1-R1. C1 and C0 are with and without conservative update technique respectively, while R1 and R0 are with and without the resetting technique. The C1-R0 configuration performs the best.

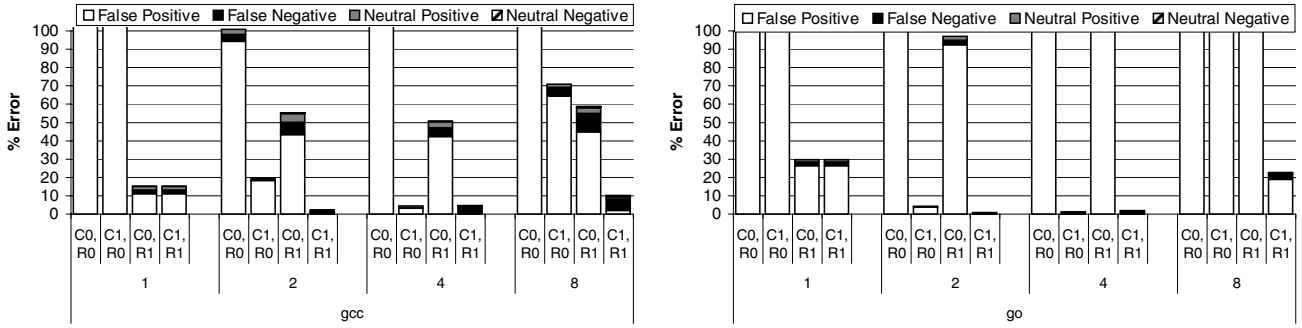

Figure 11: Muti-hash profiler for an interval of 1 million, 0.1% candidate threshold, and a total number of hash-table entries of 2K. Each graph shows results for multi-hash profiler with 1,2,4 and 8 hash-tables. For each hash-table, it shows results for 4 different configurations: C0-R0, C1-R0, C0-R1, and C1-R1. C1 and C0 are with and without conservative update technique respectively, while R1 and R0 are with and without the resetting technique. The C1-R0 configuration performs the best.

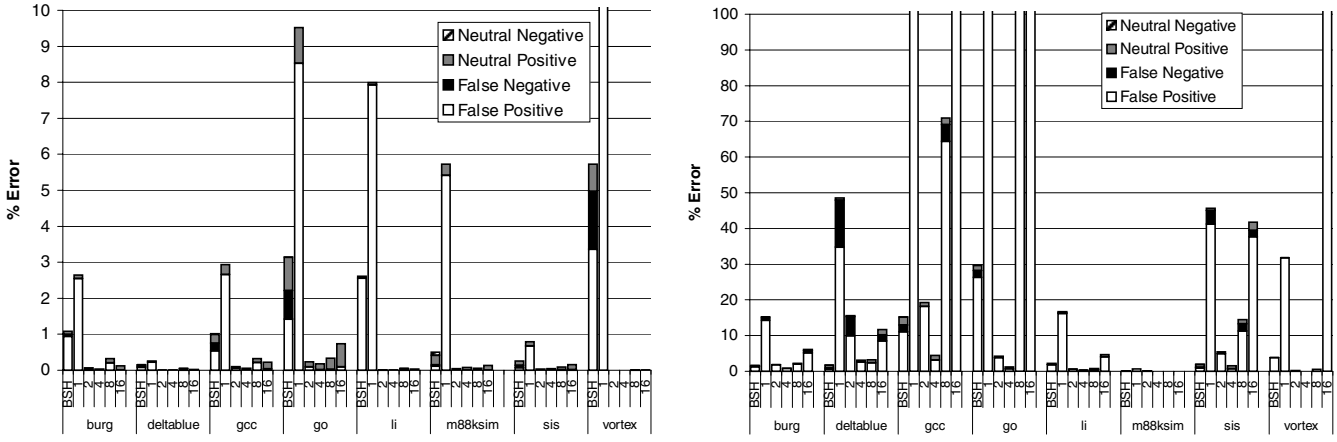

Figure 12: Best Muti-hash profiler for Value Profiling - C1, R0 (with conservative update and without resetting). For each program 1,2,4,8 and 16 hash-tables are analyzed using a total of 2K hash-table entries. The figure on the left shows results for interval 10,000 with 1% candidate threshold. The figure on the right shows results for interval 1 million with 0.1% candidate threshold. The results show that using 4 hash-tables consistently outperforms others.

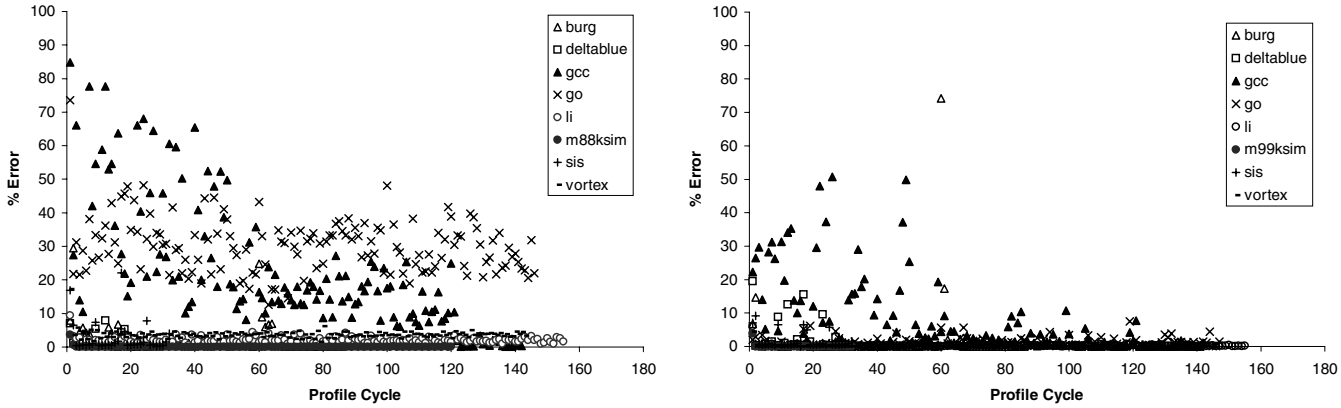

Figure 13: To explore the amount of variation in the error across various intervals we break down the error by the interval that it was gathered. This figure shows the results of that break down for an interval length of 1 million instructions with a candidate threshold of 0.1%, 2K hash table entries, and with retaining. The best single-hash architecture with resetting is shown on the left and best multi-hash configuration with 4 hash-tables, conservative update, and no resetting is shown on the right.

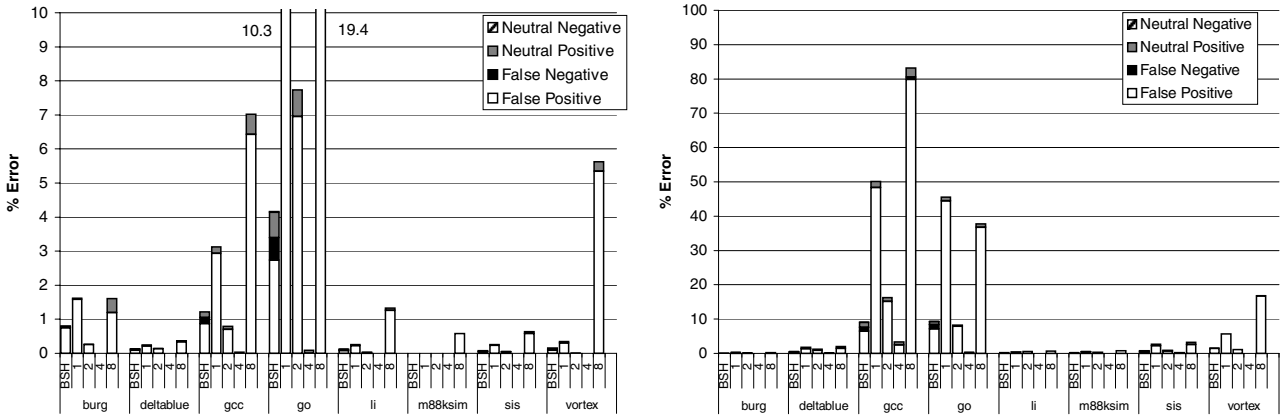

Figure 14: The best multi-hash profiler for Edge profiling. C1 R0 is used with conservative update and without resetting. For each program 1,2,4, and 8 hash-tables are analyzed using a total of 2K hash entries/counters. The left figure shows results for an interval of length 10,000 with 1% candidate threshold. The right figure shows results for an interval of length 1 million with 0.1% candidate threshold.

number of such spikes in error rates especially for `gcc` and `go`.

The one interval for `burg` that spikes with a large error rate comes from not using resetting for the multi-hash architecture. Conservative update is used instead of resetting, as described above. For this one interval, there are a large number of candidate tuples and they allow the piggy-backing of false positives into the accumulator table when using conservative update. In comparison, this spike is not seen for the single hash approach in Figure 13, because resetting prevents this effect. Even so, the average results in Figures 11 show that conservative update performs much better on average without resetting.

6.4.2 Edge Profiling

We also examined edge profiling to show that our profiler works with the same efficiency across different applications. The edge profiler will see fewer distinct tuples than value profiling. Figure 14 shows results for the best multi-hash profiler for edge profiling using 2K hash entries. It can be seen that the conclusions we made based on value profiling also hold for edge profiling. The results show that using a 4 hash-table multi-hash configuration significantly outperforms other configurations including the best single hash approach.

7. Summary

The idea of tracking program behavior in hardware tables has become one of computer architecture's most used and most effective tools for increasing performance. From branch prediction to prefetching to memory disambiguation, almost all hardware optimizations make use of some type of profiling tables or counters. In this paper, we have presented a generalized profiling architecture that has applications to both existing hardware profiling schemes and generalized profiling schemes.

By clever use of hash-tables and an accumulator table working in unison, we show how to effectively filter all of the important data from the surrounding noise, to enable the capture of the most frequently occurring events. The hash-table is used

to identify events that *may* account for a large percentage of the tuples profiled, and acts as an informant to the accumulator table, which accumulates accurate counts for candidate tuples. The accumulator table uses the information from the hash-table to further investigate the behavior of these potentially common events.

As was mentioned earlier in the paper there are several constraints that need to be met in order to allow a hardware profiling scheme to be useful. First and foremost, the act of profiling must be efficient, both in terms of hardware cost and performance overhead. We presented an architecture that requires only a small amount of area. The overall effect is that very large amounts of information can be processed while using an area proportional to the number of unique tuples that are to be examined in an interval. For the results examined in this paper, the size of the hash table was 6 Kilobytes (2K entries of 3 byte counters), and the size of the accumulator table was 1 KB for the 1% candidate threshold and 10 KB for the 0.1% candidate threshold. The performance overhead for the presented scheme is non-existent because statistics gathering is decoupled from the main execution of the program and no software or operating system interaction is needed to accumulate the data during the steady state operation of profiling.

The profile statistics gathered must be representative, accurate, and timely so that the optimizations may be applied correctly. We showed that for two potential profiling uses, value and edge profiling, these properties were met. The basic profiling architecture, when combined with the optimizations of *reset*, *conservative update*, *retaining*, and *multi-hashing*, achieved an error less than 1% on average.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF CAREER grant No. CCR-9733278, NSF grant No. CCR-0105743, NSF grant No. ANI 0074004, a grant from NIST on the Sensilla Project, a Jacobs School of Engineering fellowship, a grant from Compaq Computer Corporation, and an equipment grant from Intel.

8. REFERENCES

[1] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, R. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone. *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.

[2] M. Burrows, U. Erlingson, S. Leung, M. Vandevoorde, C. Waldspurger, K. Walker, and W. Weihl. Efficient and flexible value sampling. In *In Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 160–167, November 2000.

[3] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the 30th Annual Symposium on Microarchitecure*, December 1997.

[4] J. Collins, S. Sair, B. Calder, and D. Tullsen. Pointer-cache assisted prefetching. In *35th International Symposium on Microarchitecture*, November 2002.

[5] T. Conte, K. Menezes, and M. Hirsch. Accurate and practical profile-driven compilation using the profile buffer. In *In Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 36–45, December 1996.

[6] J. D., J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos. Profileme : Hardware support for instruction-level profiling on out-of-order processors. In *International Symposium on Microarchitecture*, pages 292–302, 1997.

[7] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *ACM SIGCOMM*, August 2002.

[8] T. Heil and J. Smith. Relational profiling: Enabling thread level parallelism in virtual machines. In *In Proc. 33rd International Symposium on Microarchitecture*, December 2000.

[9] T.H. Heil and J.E. Smith. Selective dual path execution. Technical report, University of Wisconsin-Madison, 1996.

[10] Andreas Krall. Efficient JavaVM just-in-time compilation. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, September 1998.

[11] M. Merten, A. Trick, E. Nystrom, R. Barnes, and W. Hwu. A hardware mechanism for dynamic extraction and relayout of program hot spots. In *in Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[12] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *ISCA*, pages 136–147, 1999.

[13] R. Muth, S. A. Watterson, and S. K. Debray. Code specialization based on value profiles. In *Static Analysis Symposium*, pages 340–359, 2000.

[14] E. Rotenberg, S. Bennett, and J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *29th Annual International Symposium on Microarchitecture*, December 1996.

[15] S. Sastry, R. Bodik, and J.E. Smith. Rapid profiling via stratified sampling. In *28th Annual International Symposium on Computer Architecture*, June 2001.

[16] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002. http://www.cs.ucsd.edu/users/calder/simpoint/.

[18] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[19] G. Tyson, K. Lick, and M. Farrens. Limited dual path execution. CSE-TR 346-97, University of Michigan, 1997.

[20] S. Wallace, B. Calder, and D. Tullsen. Threaded multiple path execution. In *25th International Symposium on Computer Architecture*, June 1998.

[21] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. Technical Report, Univ. of Arizona, Dept. of CS, Tucson, AZ, June 2000.

[22] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value-centric data cache design. In *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 150–159, November 2000.

[23] C. Zilles and G. Sohi. A programmable co-processor for profiling. In *In Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, January 2001.